Public, internal top-level classes :

Classes which are defined inside any namespace i.e not nested within any class are known as top-level classes. These classes can be either public or internal. Default accessibility of top-level classes is internal i.e if we do not specify any accessibility specifier while defining it.

If we define top-level class without using any accessibility specifier i.e “internal” then it can be accessed anywhere inside the namespace or assembly in which it is defined.

For eg.

```
Using system.*;

Namespace toplevelclasses

{
        Class Myclass {

        Int a;

        String b;
        }
}
```

In the above eg., Class “Myclass” is internal by default i.e no accessibility specifier. If we define a top-level class with “public” access specifier then it can be accessed anywhere inside the application (i.e inside different namespaces or assemblies).

For eg.

```
Using system.*;

Namespace toplevelclasses
{
        Public Class Myclass {
        int a;
        string b;
        }
}
```

In the above eg. Class “Myclass” is public.All the members inside the top-level classes are private by default.

Struct and enum as members of class :

Basically, structures are value types and not reference types. Structures cannot inherit other

classes or structures. They cannot be used as base for other classes or structures.

However all the structures inherit “System.Valuetype” class which inherits “’Object” class implicitly.

Structures can implement 1 or more Interfaces.but no interface can inherit structure.Enum is used when we want to provide the available option to the user from the list.Data is stored as indexes i.e 0 to ∞.

We can have both struct and enum as members of a class. By default they are private inside the class.

For eg.

```
Using system.*;
Namespace StructAndEnum
{
        Class myclass {

        Public Struct mystruct{

        Int x;

        Public int add(int a,int b) {

        x = a+b;

        return x;
        }
}

Static void main() {

myclass m1=new myclass();

mystruct struct=new struct();

m1.struct.add(10,20);

}
}

}
```

In the above eg. We defined public struct “mystruct” inside the class “myclass” which can be accessed anywhere inside the application. By default it is private.

For eg.

```
Using system.*;

Namespace StructAndEnum

{

Class enumfun {

Public enum colors {

Violet;

Indigo;

Blue;

Green;

Yellow;

Orange;

Red;

}


Static void main() {

Console.WriteLine(enumfun.colors.Violet);

}
```

In the above eg. We defined public enum “color” inside the class “enumfun” which can be accessed anywhere inside the application. By default it is private.

Base keyword:

The base keyword is used to access members of the base class from within a derived class: Call a method on the base class that has been overridden by another method.

Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the base keyword from within a static method.
Example:

```
class    TwoDShape    {
  double       pri_width;
  double pri_height;

  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }
  Public virtual double area() {
    return width * height / 2;
  }
}

class Triangle : TwoDShape {
  string style;

  public Triangle(string s, double w, double h) : base(w, h) {
    style = s;
  }
  Public override double area() {
  base.area();}
}

Class EntryPoint
{
public static void Main() {
    Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
    Console.WriteLine(t1.doublearea());
}
```

Abstract keyword:

The abstract modifier can be used with classes, methods, properties, indexers, and events.Use

the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

In this example, the class Square must provide an implementation of Area because it derives from ShapesClass:

```
abstract class ShapesClass

{
   abstract public int Area();
}

class Square : ShapesClass

{
   int x, y;

  public override int Area()

   {
      return x * y;}
}
```

Abstract classes have the following features:

An abstract class cannot be instantiated.
An abstract class may contain abstract methods and accessors.

It is not possible to modify an abstract class with the sealed (C# Reference) modifier, which means that the class cannot be inherited.
A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.
Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

An abstract method is implicitly a virtual method.
Abstract method declarations are only permitted in abstract classes.

Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature.

For example:

public abstract void MyMethod();
The implementation is provided by an overriding methodoverride (C# Reference), which is a member of a non-abstract class.
It is an error to use the static or virtual modifiers in an abstract method declaration.
Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.
It is an error to use the abstract modifier on a static property.
An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.
An abstract class must provide implementation for all interface members.
An abstract class that implements an interface might map the interface methods onto abstract methods.

For example:

```
interface I
{
void M();
}
abstract class C: I
{
   public abstract void M();
}
```

Polymorphism:

Polymorphism literally means "many forms". It adds enormous flexibility to programs. In general terms, polymorphism allows the same code to have different effects at run-time depending on the context. Polymorphism takes advantage of inheritance and interface implementation.
Method overloading supports polymorphism because it is one way that C# implements the "one interface, multiple methods" paradigm.
Polymorphism can be *static* or *dynamic*. In static polymorphism, the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.

Example:

Lets take the example of some words which can be used in different situations with different meaning. 'watch' - watch can be used as a verb as well as a noun. "He is wearing a watch." In this, it is used as a noun. "Watch your actions." In this, it is used as a verb. Thus we see that the same word can be used in different situations differently.

Encapsulation:

Encapsulation is the first pillar or principle of object-oriented programming. In simple words, "Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit" and Class is the best example of encapsuation.

Definition:

Encapsulation is the process of separating the aspects of an object into external and internal aspects. The external aspects of an object need to be visible or known, to other objects in the system. The internal aspects are details that should not affect other parts of the system. Hiding the internal aspects of an object means that they can be changed without affecting external aspects of the system.

Example:

Consider the radio in a car. The external aspects of the radio are control buttons, connectors, speakers and antenna. The internal aspects are the details about how the radio works. To use a radio in a car, you need not know anything about electrical engineering. We can replace the radio in a car with the one which has CD Player without affecting the other components of the car. Because the operation of radio has been encapsulated and external view of the radio is defined by the control buttons and the connectors.

We can achieve encapsulation by the following ways

1. By using the get and set methods
2. By using properties (read only properties, write only properties)
3. Using an Interface

"new" Keyword

1) new Operator:-used to create objects on heap and invoke constructors.

How memory get allocate using new operator?

Yes, the new operator creates objects of a specified class.

For Example:

```
class Test
  {
  }
  class Program
  {
    static void Main(string[] args)
    {
      Test t = new Test();
    }
  }
```

the new object has created, object of Test class. 'new' operator allocates memory"

But,

New operator always does not allocate memory:

Here , I have created an array and have used the new operator to allocate memory for 10 elements. The following is C# code.

```
class Program

static void Main(string[] args)
      {
         int[] a = new int[10];}}
}
```

no memory allocation has occured in spite of using the new operator. Then when will the memory be allocated? The answer is "when we use it in an array in our program". Actually the .NET code optimizer is very smart when it sees that there is no use of an array in the current program, it simply omits the array from being processed.
Here is program in which declared an array and just made a little operation on it. As in the following:

```
class Program
  {
     static void Main(string[] args)
     {
        int[] a = new
        int[10]; a[0] = 100;
     }
  }
```

A very interesting fact of the new operator:

The interesting fact of the new operator is, when we use the new operator to create an object (if the .NET code optimizer thinks it is really needed) of a particular data type then the initialization operation is also done with that .Here I have created one integer variable "a" using the new operator and without initializing it I want to print it's value.

```
class Program
  {
     static void Main(string[] args)
     {
        int a = new int();
        Console.Write("Local variable a:- "+ a
        +"\n"); Console.ReadLine() }
  }
```

Output- Local variable a:-0

See, we do not set the value 0 to the variable "a" but it's showing the value of the variable "a" is 0. But if we try without using the new operator as in the following:

```
class Program
{
    static void Main(string[] args)
    {
    int a;
        Console.Write("Local variable a:- "+ a
        +"\n"); Console.ReadLine();
    }
}
```

Error. Use of unassigned local variable "a".

```
class Test
   {
     public int Roll;
     public String
     name;
   }
   class Program
   {
      static void Main(string[] args)
      {
         int a = new int();
         Console.Write("Local variable a:- "+ a
         +"\n"); Test t = new Test();
         Console.Write("String class member:- "+ t.name + "\n");
         Console.Write("Integer class member:- " + t.Roll +"\n");
         Console.ReadLine();
      }
   }
```

Here is the output
Local variable a:-0
String class member :-
Integer class member :-0
The String class member is blank because it's null.

2)new Modifier:-used to hide an inherited member from a base class in child class.

<u>For Variable:-</u>

```
class Parent { public int i = 0;
}
// Create a derived class. class Child :
Parent {
   new int i; // here i hides the i in Parent
   public Child(int a, int b) {
      base.i = a; // here base.i uncovers the i in Parent i = b; // i in Child
   }
   public void show() {
```

```
        // this displays the i in Parent. Console.WriteLine("i in Parent class: "
        + base.i);
        // this displays the i in Child Console.WriteLine("i in derived
        class: " + i);
    }
}
```

For method:-

Consider the following example:

```
public class baseHello { public void
sayHello()
        {
                System.Console.WriteLine("base says hello");
        }
}

class derivedHello:BaseHello
{ public void sayHello()

{
System.Console.WriteLine("derived says hello");
}
}
```

The preceding code will compile fine but the compiler warns you that the method derivedHello.sayHello() hides the method baseHello.sayHello():

warning CS0114: 'derivedHello.sayHello()' hides inherited member 'baseHello.sayHello()'. To make the current method override that implementation, add the override keyword. Otherwise add the new keyword.

As the warning suggests, it is preferable to use "new" as in the following:

```
class derivedHello:BaseHello

{
new public void sayHello()

{
System.Console.WriteLine("derived says hello");

}

}
```

public

The type or member can be accessed by any other code in the same assembly or another assembly that references it. private

The type or member can only be accessed by code in the same class or struct.
protected

The type or member can only be accessed by code in the same class or struct, or in a derived class.
internal

The type or member can be accessed by any code in the same assembly, but not from another assembly.

When no access modifier is set, a default access modifier is used. So there is always some form of access modifier even if it's not set

protected internal

The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

static

The static modifier on a class means that the class cannot be instantiated, and that all of its members are static. A static member has one version regardless of how many instances of its enclosing type are created.

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be externally instantiated. In other words, you cannot use the new keyword to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself.

However, there is a such thing as a static constructor. Any class can have one of these, including static classes. They cannot be called directly & cannot have parameters (other than any type parameters on the class itself). A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.

visibility / keyword	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Checked

Use checked when you don't want *accidental* overflow / wrap-around to be a problem, and would rather see an exception.

Ref Keyword

The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.

Out Keyword

The out keyword passes arguments by reference. This is very similar to the ref keyword.

Ref Vs Out

Ref	Out
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.	A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.
Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter.	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
It is not compulsory to initialize a parameter value before using it in a calling method.	A parameter value must be initialized within the calling method before its use.
When we use REF, data can be passed bi-directionally.	When we use OUT data is passed only in a unidirectional way (from the called method to the caller method).
Both ref and out are treated differently at run time and they are treated the same at compile time.	
Properties are not variables, therefore it cannot be passed as an out or ref parameter.	

The const keyword is used to modify a declaration of a field or local variable.
It specifies that the value of the field or the local variable is constant, which means it cannot be modified
A constant expression is an expression that can be fully evaluated at compile time
Remarks

The constant declaration can declare multiple constants, for
example: public const double x = 1.0, y = 2.0, z = 3.0;

static const int x = 55; gives err as const is implicitly static.
**********Working with Constant Field Data**********
****

C# offers the const keyword to define constant data, which can never change after the initial assignment.
As you might guess, this can be helpful when you are defining a set of known values for use in your applications that are logically connected to a given class or structure.
Assume you are building a utility class named MyMathClass that needs to define a value for the value PI (which you will assume to be 3.14). Begin by creating a new Console Application project named ConstData. Given that you would not want to allow other developers to change this value in code, PI could be modeled with the following constant:

```
namespace ConstData
{
class MyMathClass
{
public const double PI = 3.14;
}
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Const *****\n");
Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
// Error! Can't change a constant!
// MyMathClass.PI          =
3.1444;
Console.ReadLine();
}
}
}
```

Notice that you are referencing the constant data defined by MyMathClass using a class name prefix (i.e., MyMathClass.PI). This is due to the fact that constant fields of a class are implicitly static. However, it is permissible to define and access a local constant variable within a type member. By way of example:

```
static void LocalConstStringVariable()
{
// A local constant data point can be directly
accessed. const string fixedStr = "Fixed string
Data"; Console.WriteLine(fixedStr);
// Error!
fixedStr = "This will not work!";
}
```

Regardless of where you define a constant piece of data, the one point to always remember is that the initial value assigned to the constant must be specified at the time you define the constant.Thus, if you were to modify your MyMathClass in such a way that

the value of PI is assigned in a class constructor as follows:

```
class MyMathClass
{
// Try to set PI in ctor? public const double PI; public MyMathClass()
{
// Error!
PI = 3.14;
}
}
```

you would receive a compile-time error. The reason for this restriction has to do with the fact the value of constant data must be known at compile time. Constructors, as you know, are invoked at runtime.
**********READ

ONLY**********
****
he readonly field can be used for run-time constants

**********Understanding Read-Only Fields**********

Closely related to constant data is the notion of read-only field data (which should not be confused with a read-only property). Like a constant, a read-only field cannot be changed after the initial assignment.
However, unlike a constant, the value assigned to a read-only field can be determined at runtime and, therefore, can legally be assigned within the scope of a constructor, but nowhere else.
This can be very helpful when you don't know the value of a field until runtime, perhaps because you need to read an external file to obtain the value, but wish to ensure that the value will not change after that point.

```
using System;

namespace ConstData
{
classMyMathClass
  {
Public readonly double PI;
```

```
public MyMathClass (double d)
      {
         PI = d;
      }
   }

classProgram
   {
staticvoid Main(string[] args)
      {
Console.WriteLine("***** Fun with Const *****\n");
// Error! Can't change a constant!
// MyMathClass.PI = 3.1444;
MyMathClass mm = newMyMathClass(3.14);
Console.WriteLine("{0}", mm.PI);
Console.ReadLine();
      }
   }
}
```

**********Static Read-Only Fields**********

Unlike a constant field, read-only fields are not implicitly static. Thus, if you want to expose PI from the class level, you must explicitly make use of the static keyword.

If you know the value of a static readonly field at compile time, the initial assignment looks very similar to that of a constant (however in this case, it would be easier to simply use the const keyword in the first place, as we are assigning the data field at the time of declaration):

```
class MyMathClass
{
public static readonly double PI = 3.14;
}
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Const *****");
Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
Console.ReadLine();
}
}
```

However, if the value of a static read-only field is not known until runtime, you must make use of a static constructoras described

```
class MyMathClass
{
public static readonly double PI;
static MyMathClass()
{ PI = 3.14; }
}
```

**********Properties***

****

Property is used to Modify the private variable of Class from out side of the code like

```
class A
{
private string var1;
public string SetVar1
{
  get{return var1;}
  set{var1 = value;}
}
}
```

now if i want to modify var1 value from class B then you need property to modify it so that you can modify it whenever you need to like below

```
class B
{
  //Modify Property
   A obj = new A();
    obj.SetVar1 = "hello";
```

here you will argue that we can modify the Var1 also by making it public so we can access it from out side but property modified it without making it public :)

Now Constructor

Constructor is Actually a Method that has same name as Class name and that method executes When you create Instance of class .

Example

```
class A
{
   int x,y;
   A(int a,int b)
  {
  x =a;
     y=b;
   }
}
```

Now here we can use Constructor to initialize the internal variable of class when we make its instance :)

```
class B
{
   A obj = new A(5,10);
}
```
so we assigned the value to x,and y when we created Instance of class :)

Constructor Vs Property

---

as you know You can Set Property whenever you want to change the Class Variable value but Constructor can not be executed after Creating Instance :) thats the difference
A class in C# is a vehicle for translating an abstraction to user defined type. It combines data representation and methods for manipulating that data into one neat package.

Classes are data types based on which objects are created with similar properties and methods are grouped together to form a class.thus a class represent a set of individual objects.
Characteristics of an object are represented in a class as properties.The actions that can be performed by objects become functions of the class and is referred to as methods.

For example consider we have a class of cars under which santro xing,alto and wagonR represents individual objects.In this context each car object will have its own model year of manufacture ,colour ,top Speed,Engine power etc.Which form properties of the Car class and the associated actions i.e Object Functions like Start,Move,Stop forms the methods of car class.
next you can define the class.Generally,a class specification has two part:

1) A class declaration, which describes the data component, in terms of member function
2)The class method definations,which describe how certain class member functions are implemented.

Sealed keyword:

Sealed is a modifier, which when applied to a class or a method of a class or a property of that class prevents other classes from inheriting from it.

In simple words consider this example.

```
Class A{}
Sealed Class B:A
{ }
```

In the above example Class B is inherited from Class A but now, Class B is modified as sealed so if you try to do something like this,

```
Class C : B
{}
```

You will get a compile time error “Cannot derive from sealed type”.
You can also use the sealed modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow to inherit but restrict overriding of those method or properties which are declared sealed.

Consider the following class

```
Class Base{
Protected virtual void fun1()
{
Console.Writeline(“base fun1”);
}
Protected virtual void fun2()
{
Console.Writeline(“base fun2”);
}
}
Class Sub : Base
{
Sealed protected override void fun1()
{
Console.Writeline(“sub fun1”);
}
protected override void fun2()
{
Console.Writeline(“sub fun2”);
}
}
```

Now if you try to inherit this class it will allow you to inherit BUT you cannot override the methods which are sealed.

```
Class Child : Sub{
Protected override void fun1()  //compile time error
{
Console.WriteLine("child fun1");
}
Protected override void fun2()  //Allowed
{
Console.WriteLine("child fun2");
}
}
```

Note that sealed keyword cannot be applied on a virtual or abstract methods
As these are opposite in nature. Hence we used sealed keyword on overriding method.
Structs are also internally sealed so they cannot be inherited.
There are two scenarios which u have to consider while making a class or a method sealed.

1. The potential benefits deriving class might gain by inheriting that class.
2. The sensitivity of your method which can be overridden by the Child class there might be a chance your overriding method might not work as expected.

Virtual keyword:

The virtual keyword is used to modify a method, property, indexer or event declaration and allow for it to be overridden in a derived class.
For example this method can be overridden by any class that is derived from it.

```
Public virtual double area(int x, int y)
{
Return x * y;
}
```

This method can be overridden by the derived class by using override keyword.

```
Public override double area(int x,int y)
{
Return (x * y)/2;
}
```

When a virtual method is invoked the run time type of object is checked for an overriding member. It is not compulsory to override the virtual method.
In C#, by default methods are non-virtual. You cannot override a non-virtual method.
You cannot use virtual keyword with static, abstract, private or override members.

STATIC

Stactic is internally sealed.static
The static keyword can be applied
to the following:

1)Data of a class
2)Methods of a
class 3)Properties
of a class 4)A
constructor

5)The entire class definition

eg:You are relaxing on a tropical island. No one else is there. A bird flies by in the sky. The island is static in memory—only one instance exists.

STATIC MEMBER
1)default value is
0 2)initialised only
once 3)persist
4)every object will have same copy

Most of the time when designing a class, you define data as instance-level data; said another way, as nonstatic data. When you define instance-level data, you know that every time you create a new object, the object maintains its own independent copy of the data. In contrast, when you define static data of a class, the memory is shared by all objects of that category

Static data, on the other hand, is allocated once and shared among all objects of the same class category.

Eg saving a/c interest 4.3 for all

Balanceamt/ instance data

Here, our assumption is that all saving accounts should have the same interest rate. Because static data is shared by all objects of the same category, if you were to change it in any way, all objects will “see” the new value the next time they access the static data, as they are all essentially looking at the same memory location.

the CLR will allocate the static data into memory exactly
one time. Static get called when a member make use of it
in entrypoint

STATIC CONSTRUCTOR

This approach will ensure the static field is assigned only once, regardless of how many objects you create. However, what if the value for your static data needed to be obtained at runtime? For example, in a typical banking application, the value of an interest rate variable would be read from a database or external file. To perform such tasks requires a method scope such as a constructor to execute the code statements. For this very reason, C# allows you to define a static constructor, which allows you to safely set the values of your static data Simply put, a static constructor is a special constructor that is an ideal place to initialize the valuesof static data when the value is not known at compile time (e.g., you need to read in the value from an external file, a database, generate a random number, or whatnot

Here are a few points of interest regarding static constructors:
1)A given class may define only a single static constructor. In other words, the static constructor cannot be overloaded.
2)A static constructor does not take an access modifier and cannot take any parameters.
3)A static constructor executes exactly one time, regardless of how many objects of the type are created.
5)The runtime invokes the static constructor when it creates an instance of the class or before accessing the first static member invoked by the caller.
6)The static constructor executes before any instance-level constructors

DEFINING STATIC CLASSES

It is also possible to apply the static keyword directly on the class level. When a class has been defined as static, it is not creatable using the new keyword, and it can contain only members or data fields marked with the static keyword. If this is not the case, you receive compiler errors.

Note Recall that a class (or structure) that only exposes static functionality is often termed a utility class. When designing a utility class, it is good practice to apply the static keyword to the class definition

example:math class
This Keyword: The this keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

The following are common uses of this:

- To qualify members hidden by similar names, for example:C#

```
public Employee(string name, string alias)
{
    // Use this to qualify the fields, name and alias:
    this.name = name;
    this.alias = alias;
}
```

- To pass an object as a parameter to other methods, for example:
- CalcTax(this);
-
- To declare indexers, for example:

C#

```
public int this[int param]
{
    get { return array[param]; } set {
    array[param] = value; }
}
```

Static member functions, because they exist at the class level and not as part of an object, do not have a this pointer. It is an error to refer to this in a static method.

Example

In this example, this is used to qualify the Employee class members, name and alias, which are hidden by similar names. It is also used to pass an object to the method CalcTax, which belongs to another class.

C#

```
class Employee
```

```
{
    private string name; private string alias;
    private decimal salary = 3000.00m;
    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias: this.name = name;
        this.alias = alias;
    }
    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias); // Passing the object to the
        CalcTax method by using this: Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }
    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}
class MainClass
{
    static void Main()
                // Create objects:
                Employee E1 = new Employee("Mingda Pan", "mpan");
                // Display results:
                E1.printEmployee();
            }
        }/*
        Output:
            Name: Mingda Pan
            Alias: mpan Taxes:
            $240.00
         */
```

InterFace:

An interface contains only the signatures of methods, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition. In the following example, class ImplementationClass must implement a method named SampleMethod that has no parameters and returns void.

For more information and examples, see Interfaces.

Example

C#

```
interface ISampleInterface
{
    void SampleMethod();
}
class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation: void
    ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }
    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();
        // Call the member.
        obj.SampleMethod();
    }
}
```

An interface can be a member of a namespace or a class and can contain signatures of the following members:

- Methods
- Properties
- Indexers
- Events

An interface can inherit from one or more base interfaces.
When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

For more details and code examples on explicit interface implementation, see

Explicit Interface Implementation.

Example

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements IPoint has integer properties x and y.

C#

```
interface IPoint
{
  // Property signatures: int
  {
    get; set;
  }
  int y
  {
    get; set;
  }
}
class Point : IPoint
{
  //  Fields: private int _x; private
  int _y;
  //  Constructor:
  public Point(int x, int y)
  {
    _x = x; _y = y;
  }
  // Property implementation: public int x
  {
    Get
    {
      return _x;
    }
  set
  {
    _x = value;
  }
}
public int y
{
  get
  {
    return _y;
  }
  set
  {
    _y = value;
  }
}
```

```
  }
class MainClass
{
  static void PrintPoint(IPoint p)
  {
    Console.WriteLine("x={0}, y={1}", p.x, p.y);
  }
  static void Main()
  {
    Point p = new Point(2, 3);
    Console.Write("My Point: ");
    PrintPoint(p);
  }
}
// Output: My Point: x=2, y=3
```